causes the second thread to "terminate execution", while Claim 7 has been amended to recite that the quiesce event causes the second thread to "suspend execution".

Claim 8 has been amended to refer to the "first thread" recited in Claim 1. The "control" referred to in Claim 8 is program execution control (as determined by the contents of the program counter); compare the similar language in the Anschuetz patent et al. at col. 4, lines 31-33. This language is conventional in the art, and no further amendment of Claim 8 is believed to be necessary.

Claims 9-11 have been amended to refer to a "resource required by another thread", rather than to a "critical resource" as before. The steps for determining whether any such resources are held are not recited, since they are conventional (e.g., checking lock table entries) and do not form part of the present invention.

Finally, Claims 13 and 14 have been amended to refer to the suspension event recited in Claim 12.

Claims 1-4 and 6-14 amended are believed to comply fully with 35 U.S.C. § 112, second paragraph, in view of the changes discussed above.

Claims 1-4 and 6-14 are further believed to distinguish patentably over the prior art. The Examiner's rejection of these claims under 35 U.S.C. § 103 as being unpatentable over Jackson 5,297,274 in view of Anschuetz et al. 5,305,455 is therefore respectfully traversed.

Jackson discloses a performance analysis system in which a monitor application 32 inserts within the memory space 34 of a

selected target application 30 a running thread program ("mole" program) 36 which generates breakpoint interrupts on a periodic basis. Each time a breakpoint interrupt is generated by the running thread program 36, program execution of the selected target application 30 is temporarily suspended so that the monitor application 32 may analyze the current state of the registers within the application to determine where execution is taking place, after which execution of the target application is resumed.

Anschuetz et al. discloses a system for managing exceptions on a per thread basis in a multitasking, multithreaded operating system. When a process termination exception occurs, each thread of the process is accessed to execute any process termination exception handler associated with the thread.

The Examiner contends that it would have been obvious to combine the teachings of Jackson with the per thread exception management of Anschuetz et al. "for the purpose of delivering process termination exception to each thread of the process that is terminating and accessing each thread to execute any process termination exception handler associated with the thread". This argument is untenable.

In applicants' claimed system, the _invoking thread_ (i.e., the recited "first thread" that detects the application event) is suspended until the target threads (i.e., the second thread and any other additional threads) have quiesced in response to the quiesce event sent to those threads. This is to ensure that the target threads have quiesced in a controlled manner (and have released any critical resources) before the invoking thread attempts any action that might result in a deadlock. Moreover, the suspension and subsequent resumption of the invoking thread occur _before_ the application event is processed.

Jackson teaches the concept of suspending and then resuming a target application 30 (without differentiating among threads) in response to the detection of a breakpoint event by an invoking thread 36. There is no teaching of suspending execution of a first thread until a second thread has quiesced in response to a quiesce event sent to that thread, nor of resuming execution of the first thread to process an application event when the second thread has suspended in response to the suspension event sent to that thread, as recited in Claims 1 and 12 as amended. On the contrary, the only resumption of execution taught in this reference is performed <u>after</u> the breakpoint events are processed.

Anschuetz et al., with its per thread exception management, does not cure this deficiency in the teachings of Jackson. As noted by the Examiner, when the exception in question is a process termination exception, the other threads are terminated using process termination exception handlers associated with the threads. This has no apparent relation to Jackson, which is concerned with suspending an application rather than terminating it. In any event, even if the teachings of Anschuetz et al. were applied in the manner suggested by the Examiner, there would still be no teaching of suspending execution of a first thread until a second thread has quiesced in response to a quiesce event sent to it, as claimed by applicants.

In sum, the cited references simply do not consider the problem of coordinating exception handling by threads sharing common resources so as to avoid deadlocks.

This inattention is not surprising given the differences in environments. In the cited references, the basic operating system (OS/2) consists of a noninterruptible kernel in a uniprocessor environment where there are no critical resources obtained in the run-time library (RTL). By contrast, the

mainframe MVS environment consists of an interruptible kernel typically running on multiple processors with a RTL that does obtain critical resources. The significance of these distinctions is as follows.

When running multiple threads in the uniprocessor environment of Jackson and Anschuetz et al., only one thread can be executing at a given time. Further, when a thread calls a kernel function, the kernel cannot be interrupted to dispatch another thread. Only after the kernel completes the work for a thread and releases any critical resources can the kernel then dispatch another thread. Therefore, if a thread generates an interrupt which requires all the other threads to be stopped or terminated, the kernel is guaranteed that none of the other threads can be holding a critical resource. It is therefore possible to run the thread cleanup routines one by one.

On the other hand, when running multiple threads in a mainframe multiprocessor environment, it is possible to have multiple threads executing concurrently on as many processors. In addition to the executing threads, it is possible to have other threads which were interrupted and not yet dispatched. At any moment in time there can be multiple executing threads and interrupted threads, any or all of which may own a critical resource. When an event occurs which requires all threads to be stopped or terminated, it is necessary to first get all threads to a state where they no longer own a critical resource. If this is not done, any attempt to run cleanup handlers or stop multiple threads will frequently result in a deadlock.

For the foregoing reasons, Claims 1-4 and 6-14 as amended are believed to distinguish patentably over the art cited by the Examiner.

Claims 9-11 are further believed to distinguish over the art cited by virtue of their recitation that the second thread is quiesced only if it is determined that it is not holding any resource required by another thread. Neither of the references cited by the Examiner teaches this aspect of applicants' claimed system.

                              Respectfully submitted,

                              DONALD F. AULT, et al.

        BY:     _____
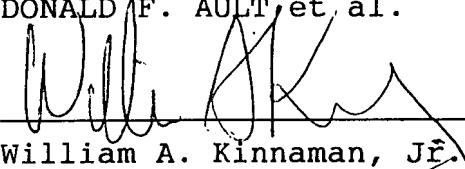
                              William A. Kinnaman, Jr.
                              Registration No. 27,650
                              Telephone:  (914)433-1175
                              FAX:        (914)432-9601